Partial Computation with a Dataflow Machine

AUTHOR(S):

ONO, Satoshi; TAKAHASHI, Naohisa; AMAMIYA, Makoto

CITATION:

ONO, Satoshi ...[et al]. Partial Computation with a Dataflow Machine. 数理解析研究所講究録 1984, 511: 169-203

ISSUE DATE:

1984-02

URL:

http://hdl.handle.net/2433/98323

数理解析研究所講究録
511巻 1984年 169-203

Partial Computation with a Dataflow Machine
データフローマシンにおける部分計算

Satoshi ONO, Naohisa TAKAHASHI and Makoto AMAMIYA
小野　諭・高橋　直久・雨宮　真人

Musashino Electrical Communication Laboratory
Nippon Telegraph and Telephone Public Corporation
3-9-11 Midoricho Musashino-shi Tokyo 180 Japan
日本電信電話公社　武蔵野電気通信研究所
(〒180　東京都武蔵野市緑町3－9－11)

Abstract

This paper presents a new dataflow computation model, called Generation Bridging Operator (GBO) model. The main features of the GBO model are as follows:

(1) The use of a partially ordered color set, called a tree structured set, as well as newly defined firing rules extended from those of the colored token (CT) model.

(2) The ability to process a closure (a pair composed of a function and an environment), which is essential to higher-order function evaluation. This model also has computational power for partial computation.

This paper also discusses a category of the GBO model named Dynamic Coloring Static Bridging (DCSB) model, in terms of its ability and limitation with regard to closure processing, partial application and partial simplification. Furthermore, this paper clarifies a dataflow graph generation method for the DCSB model by describing the main differences in code generation between the DCSB model and the CT model.

The proposed dataflow models are promising for applicative programing language machine architecture.

1. Introduction

Applicative programming languages[1-3] have various attractive features wherein they facilitate writing short and clear programs, as well as understanding and verifying these programs using clean mathematical semantics. It has also been pointed out that it is easier to detect parallelism in an applicative (or functional) program than in an imperative one [13]. From these points of view, many researchers have studied both dataflow machines[4-7] and parallel reduction machines[8,9] to find ways to execute applicative programs efficiently.

Although several promising dataflow machines have been actually implemented, there are still some problems to be solved.

(1) They don't have the ability to process a closure[10] (or funarg[11]) which is a pair composed of a function and an environment in which the function is to be evaluated.

(2) They cannot implement a partial computation[12] mechanism.

The closure concept, however, plays an important part in higher-order function evaluation, while partial computation can reduce redundant computation. To overcome the above-mentioned problems, a new dataflow computation model called Generation Bridging Operator (GBO) model, is proposed. The main features of the GBO model are as follows:

(1) It uses a partially ordered color set, called a tree structured set, as well as newly defined firing rules extended from those of the colored token (CT) model[4].

(2) This model can process a closure, and has a partial computation ability.

In the following sections, the GBO model is first defined, and compared with the CT model. Next, the GBO model is classified into four categories in terms of how to assign a color and how to determine an attribute of a dataflow graph node. After defining an applicative language, the dynamic coloring static bridging (DCSB) model, which is one type of the GBO model, is discussed in detail from the viewpoints of function application, closure

processing and partial computation. Finally, a dataflow graph generation method for the DCSB model is clarified by describing the main differences in code generation between the DCSB model and the CT model.

2. Generation Bridging Operator Model

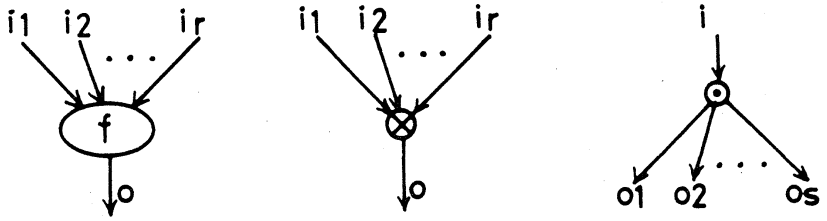
In this section, computation rules for the GBO model are defined. First, a conventional dataflow model using a colored token mechanism is described. Next, basic terms used in this paper are defined. Finally, the computation mechanism based on the GBO model is discussed.

2.1 Dataflow Model

(1) Dataflow graph

In a dataflow model, programs are represented by means of graphs consisting of nodes and arcs. A node represents an operation and an arc defines the data dependency between two nodes. Each node is connected to input arcs i1, i2, ..., ir and output arcs o1, o2, ..., os. If a node is fired (or executed), a token, which has a result value and a destination, is generated and sent to its destined nodes through its output arcs. Tokens on the input arc(s) of the node are removed.

As shown in Fig. 1, nodes are classified into three categories according to the relationship between the input and output arcs ; operation nodes, merge nodes and distribution nodes. An operation node is fired when it has tokens on all its input arcs, and then, sends a result token to its output arc. A merge node receives a token from one of its input arcs, and sends the token to its output arc. A distribution node sends a separate copy of an input token to each output arc. In order to simplify the following discussion, an operation node is assumed to have either one or two input arc(s). An output arc is called a static arc, if its destination is determined during compilation time. On the



Operation node Merge node Distribution node

Fig. 1. Dataflow nodes

other hand, it is called a dynamic arc, if its destination is determined as a result of the operation.

(2) The CT model

Every token has a result value, together with its destination and a color. The color of a token represents its execution environment. A color is often called a label, and assigning a color to a token makes it possible for more than one token to be traveling on an arc at the same time [4,5]. When a token has a value "x" and a color "c", it is represented as [x]c. (Note: in the following figures, [x] c is used instead of [x]c .)

The principal rules of firing based on the CT model are defined as follows:

[Firing rules for CT model]

(a) An operation node having only one input arc, a merge node and a distribution node are immediately firable, when an input token arrives at the node.
(b) Assume that "x" and "y" are arbitrary values and that "a" is an arbitrary color. An operation node, having two input arcs "arc1" (left-side arc) and "arc2" (right-side arc), is firable if and only if two tokens [x]a and [y]a are available on "arc1" and "arc2", respectively.
(c) When the node described in (b) is fired, tokens [x]a and [y]a are removed from the input arcs. Normally, a result token with color "a" is sent to its output arc, unless the node is a special node for setting a color (e.g. "Link" instruction node described in Section 3.2).

An example of a firing in the CT model is shown in Fig. 2. In this figure, "a", "b" and "c" are different colors. The two input tokens both having color "a" (i.e. [x]a and [y]a) are available for operation "f". Therefore, "f" is fired using input values "x" and "y". The result token [f(x,y)]a is sent to the output arc. On the other hand, since "f" gets only one token for each of the colors "b" and "c", "f" cannot yet be fired in either execution environment.

Note that there is no interaction between the tokens having different colors. Therefore, one dataflow graph can be shared among multiple execution environments.

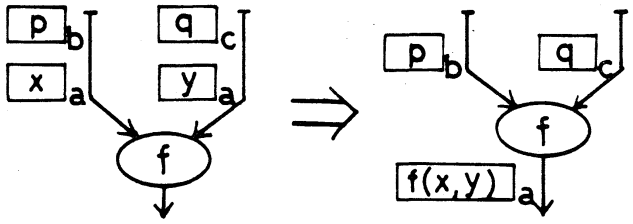


Fig. 2. Colored token (CT) model

2.2 Tree Structured Set

In this section, the notion of a tree structured set is defined and its features are discussed. First, some terms used in this paper are defined.

[Definition 1] Relation $\geqslant$

The symbol $\geqslant$ represents the partial order relation between two elements of C. That is;

$$\text{for all } a,b,c \in C$$
$$a \geqslant a$$
$$a \geqslant b \ \wedge \ b \geqslant c \ \rightarrow \ a \geqslant c$$
$$a \geqslant b \ \wedge \ b \geqslant a \ \rightarrow \ a = b$$

[Definition 2] Relation >

If "a" and "b" are elements of "C",

$$a > b \ \triangleq \ (a \geqslant b) \ \wedge \ \text{not}(a = b)$$

[Definition 3] Relation $\otimes$

We define the new relation $\otimes$ as follows:

$a \otimes b$ iff $a > b$ and there are no $x \in C$ such that $a > x > b$

This condition states that "a" is prime over "b" (or "b" is prime under "a").

[Definition 4] Composition series

The series of ai ($ai \in C$, $i=0,1,..,k$) is a composition series from "a" to "b", and "k" is called its length if and only if there exists a series such that

$$a = a0 \otimes a1 \otimes \ldots \otimes ak = b$$

[Definition 5] Ancestor and descendant

If there exists a composition series from "a" to "b", whose length is "n", then "a" is an ancestor of "b" by "n" generations (and "b" is a descendant of "a" by "n" generations).

[Definition 6] Tree structured set

The set "C" is a tree structured set, if and only if every element of "C" has at most one element which is its ancestor by one generation. (Note that this definition does not assume the existence of the maximum element in set "C".)

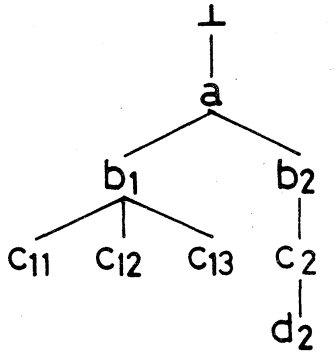


Fig. 3. Tree structured set example

An example of a tree structured set is shown in Fig. 3. In this figure, $c2 \otimes d2$ is true, while $b2 \otimes d2$ is false. Element "c11" is a descendant of element "a" by 2 generations, and element $\perp$ is an ancestor of element "d2" by 4 generations. For a tree structured set, we can immediately get the following theorem.

[Theorem 1]
Assume "C" is a tree structured set, and "a" and "b" are elements of "C", such that there exists a composition series from "a" to "b" whose length is "n". Then, "a" is a unique ancestor of "b" by "n" generations.

[Lemma 1]
Assume that "C" is a tree structured set, and that "a" and "b" are elements of "C". Then, $a \geqslant b$ if and only if there exists a unique composition series from "a" to "b".

[Lemma 2]
Assume $a \geqslant b \geqslant c$ $(a,b,c \in C)$ and that "p" is the length of the composition series from "a" to "c", and that "q" is the length of the composition series from "a" to "b". Then, there exists a unique composition series from "b" to "c" and its length is "p-q".

2.3 GBO Model Definition

In the GBO model, an operation node has a special arc tagged with an integer "n" between two input arcs, as shown in Fig. 4. The special arc and the integer "n" are called a bridge and generation difference, respectively. To simplify the following discussion without the loss of generality, it is assumed that $n \geqslant 0$. If $n < 0$, the direction of the bridge is reversed and the generation difference is changed from "n" to "-n". If $n = 0$, the bridge can be omitted and the node is the same as that for the CT model.

As in the case of the CT model, every token has a color, representing its execution environment. In the case of the GBO model, however, a set of token colors is a tree structured set. The firing rules of the GBO model for two input operation nodes are modified from those of the CT model as follows:

[Firing rules for GBO model]

An operation node, having two input arcs "arc1", "arc2" and a

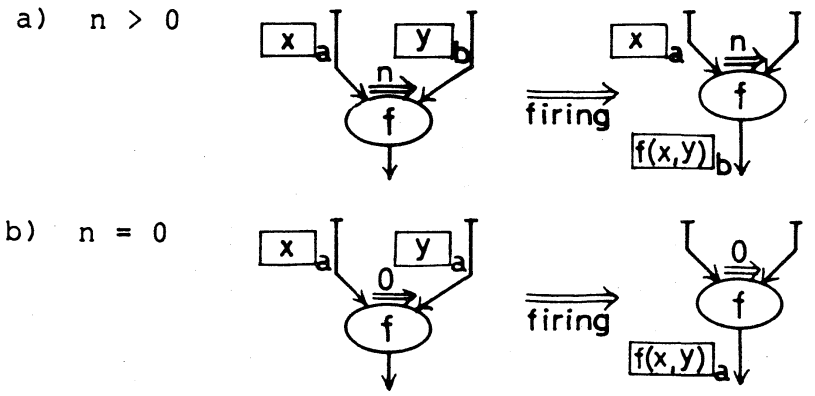


Fig. 4. Generation bridging operator (GBO) model

bridge from "arc1" to "arc2", is firable if and only if

(1) Two tokens [x]a and [y]b are available on "arc1" and "arc2", respectively,

(2) A composition series from "a" to "b" exists and its length equals the generation difference for the bridge.

As shown in Fig. 4, if the operation node "f" is fired, a result token [f(x,y)]b is sent to its output arc, except the case the node "f" itself computes the result color. Unless the length of the composition series from "a" to "b" is 0, [y]b is removed from the input arc while [x]a remains on the arc. If the length is 0, both [x]a and [y]b are removed.

If a node has a bridge from "arc2" to "arc1", assume that [x]a is on "arc2", and that [y]b is on "arc1" instead of (1) above. Then, the same firing rules apply.

From Theorem 1, color "a" can be uniquely determined, if color "b" and integer "n" are given. From Lemma 1, an integer "n" can be uniquely determined, if colors "a" and "b" ($a \geq b$) are given. Lemma 2 insures the firability for a multiple node dataflow graph.

2.4 Comparison between the CT Model and GBO Model

Given the expression

$$w = x + y + y*z$$

If "x" and "y" are fixed to 2 and 3, respectively, "w" is simplified to the following expression "w23".

$$w23 = 5 + 3*z$$

Further simplification is possible, if "z" is fixed. Using the CT model, there is no interaction between tokens having different colors, as described in Section 2.1. Therefore, the CT model can simplify "w23" only if the color of "z" is identical to the color of value tokens "x" and "y". This simplification process using "z" is destructive, however, since "w23" cannot be reused with another "z". In other words, the simplified expression "w23" cannot be shared among multiple execution environments.

On the other hand, the GBO models make it possible to share "w23" among multiple execution environments, as follows:

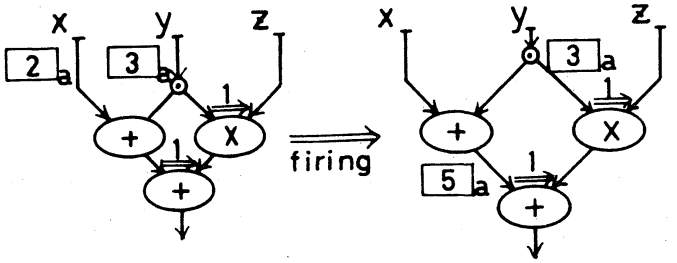
(1) If tokens [2]a and [3]a are available, expression "w" is simplified to expression

$$w23 = 5 + 3*z$$

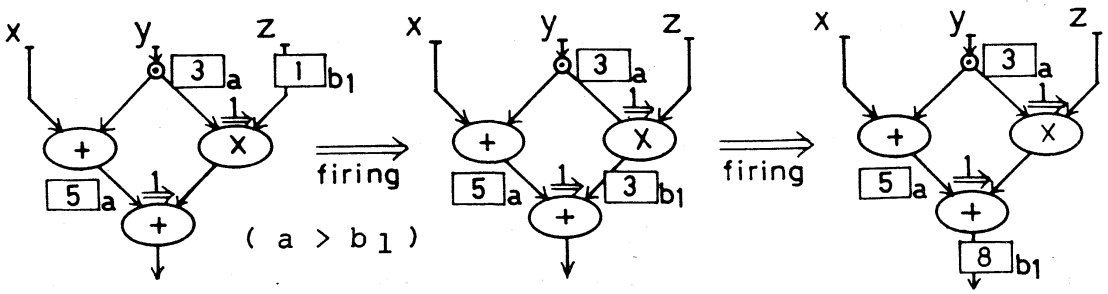
whose color is "a", as shown in Fig. 5-a).

(2) Assume that colors "bi" (i=1,2,...) satisfy bi $\otimes$ a (i=1,2,...). Expression "w23" can be concurrently simplified using multiple "z" tokens, which have different colors "bi" (i=1,2,...). For example, "w23" is simplified to [8]b1, if the input token [1]b1 arrives at the input arc of the multiply operation node, as shown in Fig. 5-b).

The capability to share a simplified dataflow graph is computational power not found in the CT model. The computational power of the GBO model is discussed in more detail in Section 3.



a) Computation when "x" and "y" are defined.



b) Computation when "z" is defined.

Fig. 5. The GBO model computation example

2.5 Classification of the GBO Models

From the viewpoints of coloring and bridging methods, the GBO models can be classified into the following four categories.

SCSB	Static Coloring	- Static Bridging
SCDB	Static Coloring	- Dynamic Bridging
DCSB	Dynamic Coloring	- Static Bridging
DCDB	Dynamic Coloring	- Dynamic Bridging

The coloring methods specify how to assign a color to an execution environment and are divided into two categories; static and dynamic. A coloring method is called a static coloring, if all colors are assigned (or scheduled to be assigned) before execution [6]. On the other hand, a coloring method is called a dynamic coloring, if color allocation is possible during execution (e.g. at function application) time [7]. It is necessary to use a dynamic coloring method to allow for recursive functions.

The bridging methods specify how to determine the generation

difference on a bridge. These are also divided into two categories; static and dynamic. The dynamic bridging method permits determination of the generation difference on bridges during execution, while the static bridging method requires that the generation difference on bridges be determined at compilation time. As shown in the following section, static bridging imposes certain restriction on function sharing.

From a practical point of view, the DCSB model appears more attractive, and is therefore discussed in more detail. A brief comparison is also made between the DCSB and DCDB models.

3. Function Application with Dataflow Model

In this section, the computational power of the DCSB model is discussed. The concepts of closure, partial application and partial simplification are considered. The relations between programs written in an applicative programming language and corresponding DCSB model dataflow graphs are shown. Then, DCSB model ability and limitation are discussed, and some comments on the DCDB model are made.

3.1 Applicative Programming Language

This section introduces an applicative programming language which is an extension of VALID[3].

The factorial function can be defined as follows:

```
fact:function = ^[[n] if n=0 then 1 else n * fact(n-1) fi ]
```

where a string ":function" indicates the data type of the symbol "fact" and may be omitted. "Fact" is a function name, and the right-side of the equation "^[[n] if ... fi]" is a function definition. "N" is a formal parameter, and "if ... fi" is a function body.

Computation is the combination of function applications and

simplifications. A function application replaces the function name with its definition, and substitutes actual parameters for formal parameters. It corresponds to the beta reduction rule[11]. Some functions are primitive and defined as axioms. Replacing a primitive function application with its result value, is called a simplification. In the following, infix operators such as "+", "*" as well as "if-then-else-fi" are assumed to be primitive.

For example, computation of fact(3) are shown

```
fact(3) = ^[[n] if n=0 then 1 else n * fact(n-1) fi ](3)
        = (if 3=0 then 1 else 3 * fact(3-1) fi)
                        :
        = (3 * fact(2))
        = (3 * ^[[n] if n=0 then 1 else n * fact(n-1) fi](2))
        = (3 * (if 2=0 then 1 else 2 * fact(2-1) fi))
                        :
        = (3 * (2 * (fact(1))))
                        :
        = (3 * (2 * (1 * (1))))
        = 6
```

Local value definitions can be used in a block enclosed by "{" and "}". The local definition equates its left-side identifier to its right-side expression.

For example, all of the following function definitions are equivalent.

```
poly = ^[[x] { y=x-1 ; z=y**2 + 2*y + 3 ; return z } ]

poly = ^[[x] { z=(x-1)**2 + 2*(x-1) + 3 ; return z } ]

poly = ^[[x] (x-1)**2 + 2*(x-1) + 3 ]
```

Identifiers defined in the block are named bound variables as are the formal parameters in the function body. The scope of the local value definition is within the block. The return value of the block is specified by the return expression[3]. This language uses static binding (lexical binding)[11]. Therefore, a free variable in the block is bound to the formal parameter of the surrounding function definition or to the value definition of

the surrounding block.

For example, assume

```
f = ^[[x] {y=(x-1)**2 ; g= ^[[z] y+2*(x-1)+z] ; return g(3)}]
```

Variables "x" and "y" in the function definition of "g" are free variables, whereas "z" is a bound variable. Variable "y" is bound to the value definition in the surrounding block, and variable "x" is bound to the formal parameter of function "f".

Computation of f(2) are shown below (unnecessary parentheses are omitted).

```
f(2) = ^[[x] {y=(x-1)**2 ; g= ^[[z] y+2*(x-1)+z] ;
               return g(3)}] (2)
     = {y=(2-1)**2 ; g= ^[[z] y+2*(2-1)+z] ; return g(3)}
     = {y=1 ; g= ^[[z] y+2+z] ; return g(3)}
     = {y=1 ; g= ^[[z] 3+z] ; return g(3)}
     = {y=1 ; g= ^[[z] 3+z] ; return ^[[z] 3+z](3)}
     = {y=1 ; g= ^[[z] 3+z] ; return 3+3}
     = 6
```

3.2 Function Application with the CT Model

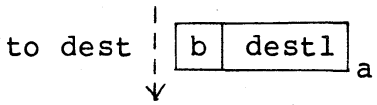
A function application is accomplished in the following way.

(1) A new color is obtained from the free color pool.
(2) Actual parameters to the function are given the color allocated in (1). Then, they are sent to the function body.
(3) The function is computed with the color in (1).
(4) The result is given the color with which the application is accomplished.

Figure 6 shows the instructions used for the function application. Three instructions i.e. "call", "link" and "rlink" (reverse link) are defined. For convenience, we give names "arg1", "arg2" and "result" to tokens on input arcs "arc1", "arc2" and on an output arc, respectively. In the figure, symbols "a" and "b" represent colors, and symbols "dest" and "dest1" are the destination node names. A pair of a color and a destination name is called an activated object name. We assume

that the value field of the token can contain an activated object name.

The symbol



means that the token having both color "a" and a value representing an activated object name is sent to the destination specified by "dest". The activated object name value is a pair of the color "b" and the destination node name "destl". We refer to the token field as:

```
tname.value.color = b
tname.value.name  = destl
tname.color       = a
tname.destination = dest
```

where "tname" stands for a token name.

The destination is specified only for the token on the dynamic arc.

The "call" instruction allocates a new color and inserts it into the color field of the activated object name. The "link" instruction accepts two tokens. It actually sends the first token to the activated object specified by the second token. The "rlink" instruction is the complement of the "link" instruction. It sends the linkage information to the specified activated object. This linkage can be used to send the result tokens back to the caller.

A dataflow graph for a function application is shown in Fig. 7. In the figure, "fname" accepts the token [* | f]a where "*" symbol indicates the "don't care" condition.

The symbols "x1", "x2",..., "xm" are actual parameters to the function "f", while "y1", "y2",..., "yn" are the destination node names for the result tokens. The new notations introduced in this figure are explained in Figs. 8 and 9. The three dataflow graphs in Fig. 8 are identical in meaning. In Fig. 9, the operation node a), is an abbreviation of the dataflow graph b). The notation #y1 stands for the activated object name constant

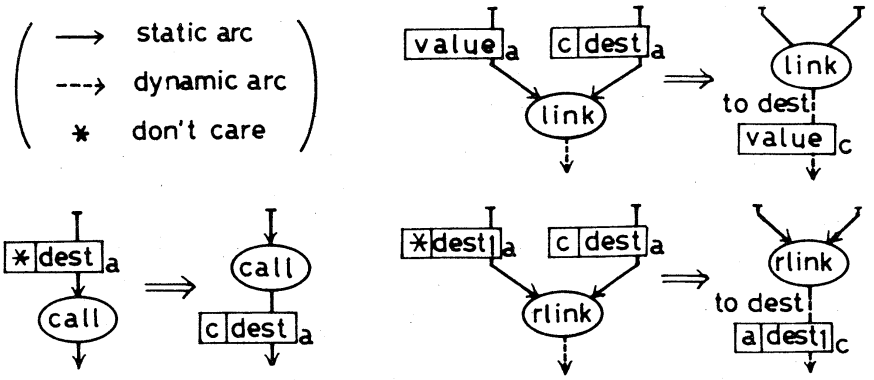


Fig. 6. Instructions for the CT model function application

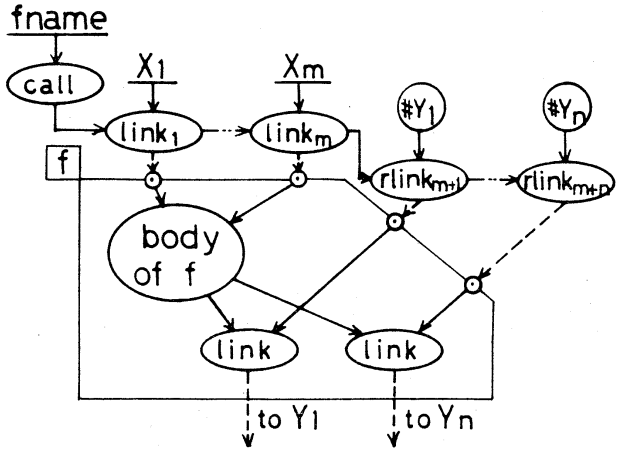


Fig. 7. Dataflow graph for a function application

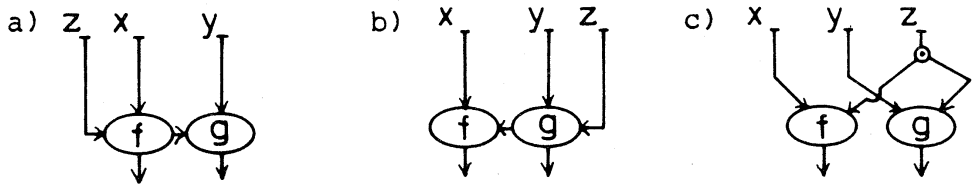


Fig. 8. Modified notations for operator nodes

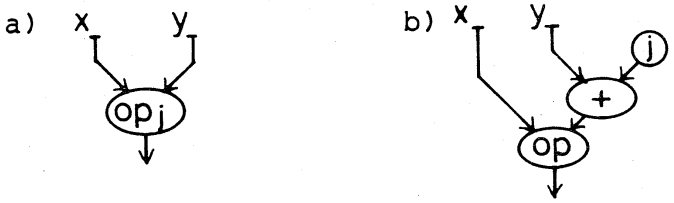


Fig. 9. Abbreviated notation for operator nodes

[⊥ | y1].

As described in Section 2.1, there is no interaction between tokens which have different colors. This enables dataflow graphs of a function to be shared by tokens having different colors. It is impossible, however, to share partially computed graphs among several colors. Therefore, the CT model cannot process a closure (or funarg) which is a pair composed of a function and an environment in which the function is to be evaluated. The notion of closure plays a very important role in functional programming languages, and therefore, the computational power of the CT model is considered insufficient.

The next section discusses the function application in the DCSB model, which allows implementation of the closure concept.

3.3 Function Application with the DCSB Model

For function applications with the DCSB model, three instructions, "call", "link" and "rlink" are extended from those for the CT model, and two new instructions are added. These are shown in Fig. 10. In the figure, it is assumed that there exists a composition series from color "a" to color "b" and that the length of the series is "n" (n > 0).

The "call" instruction accepts an activated object name as an input token. If arg1.value.color = c , the "call" instruction allocates a new color "d" which is prime under "c", and the color field of the activated object name is replaced with "d". This instruction preserves the property of the tree structured set defined in Section 2.2. The "link" and "rlink" instructions are slightly modified from those for the CT model to reflect the GBO model firing rules. The "rlink" instruction outputs the following result token.

```
result.value.color = min (arg1.color, arg2.color)
result.value.name  = arg1.value.name
result.color       = arg2.value.color
result.destination = arg2.value.name
```

The "send" instruction is similar to the "link" instruction

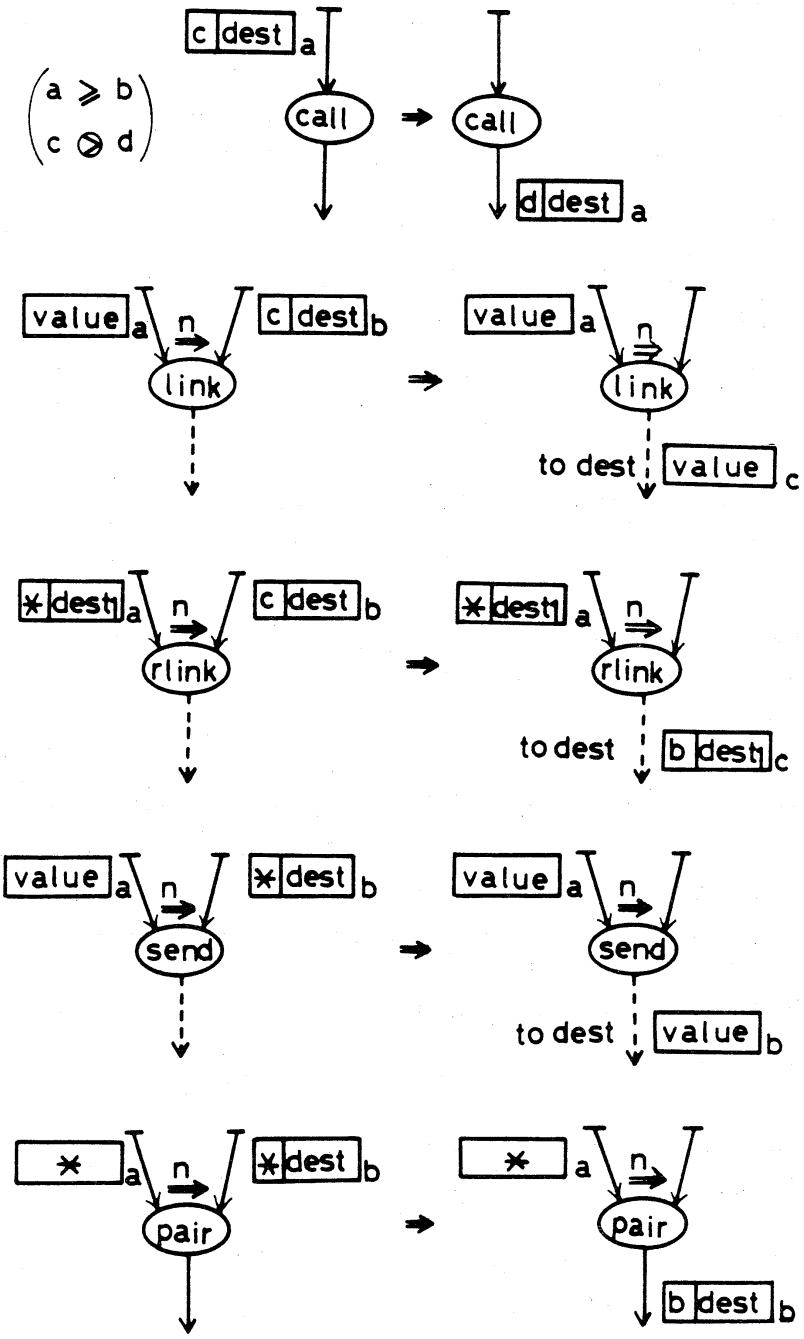


Fig. 10. Instructions for the GBO model function application

except that color re-assignment is not performed. The "pair" instruction creates an activated object name from the current environment as follows:

```
result.value.color = min (arg1.color, arg2.color)
result.value.name  = arg2.value.name
result.color       = min (arg1.color, arg2.color)
```

Using these extended instructions, the function application sequence for the DCSB model is identical to that for the CT model shown in Fig. 7. In the figure, "fname" stands for an activated object name.

3.4 Closure Processing with the DCSB Model

The DCSB model is a generalization of the CT model, and has at least the same computational power. One of the additional power resulting from this generalization is the ability to process a closure.

In the DCSB model, a closure is represented by an activated object name. A set of colors having total ordering is used to specify the environment of the function. The color field of the activated object name is the minimum color of this set.

Assume that function "f" is defined as follows:

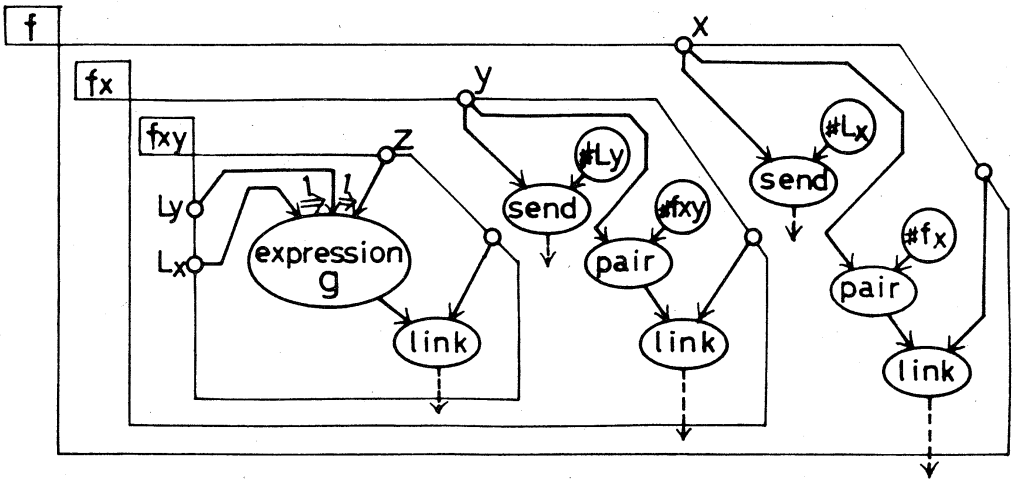
```
f = ^[[x] ^[[y] ^[[z] g(x,y,z) ] ] ]
```

where "g(x,y,z)" represents some expression of "x", "y" and "z" such as "x+y+z".

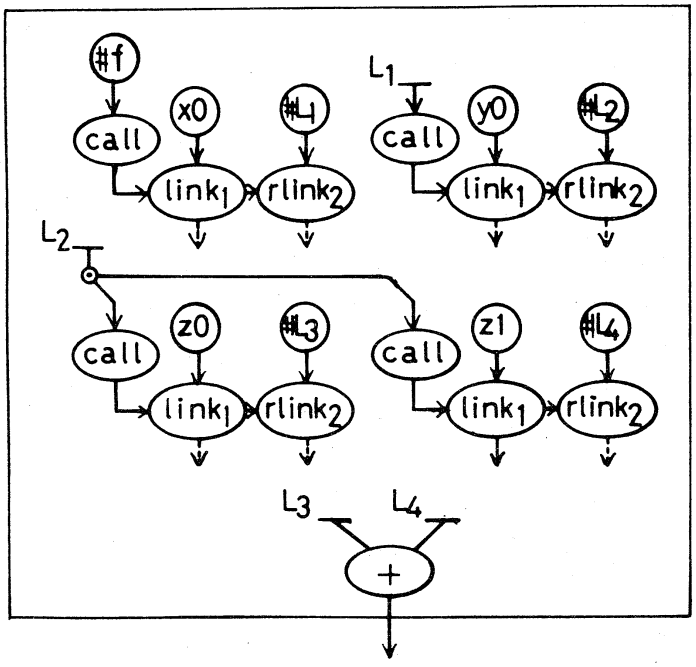
Identifier "z" in expression "g(x,y,z)" is a bound variable in the expression ^[[z] g(x,y,z)]. However, "y" and "x" are free variables and are bound by the surrounding function definition. The dataflow graph for "f" is shown in Fig. 11 a).

Now, consider the evaluation of the following block where "x0", "y0", "z0" and "z1" are constants. A corresponding graph is shown in Fig. 11 b).

```
{ fx = f(x0) ; fxy = fx(y0) ; return fxy(z0)+fxy(z1) }
```



a) Dataflow graph for function definition "f",
where f = ^[[x] ^[[y] ^[[z] g(x,y,z)]]]



b) Dataflow graph for expression
{ fx = f(x0) ; fxy = fx(y0) ; return fxy(z0)+fxy(z1) }

Fig. 11. Dataflow graphs for closure computation with the DCSB model

```
fx  = f(x0)
    = ^[[x] ^[[y] ^[[z] g(x,y,z) ] ] ] (x0)
    = ^[[y] ^[[z] g(x0,y,z) ] ]
```

Thus,

```
fxy = fx(y0)
    = ^[[y] ^[[z] g(x0,y,z) ] ] (y0)
    = ^[[z] g(x0,y0,z) ]
```

Therefore,

```
xy(z0)+fxy(z1) =  ^[[z] g(x0,y0,z) ] (z0)
                + ^[[z] g(x0,y0,z) ] (z1)
               = g(x0,y0,z0) + g(x0,y0,z1)
```

Note that the value sent back to node "L1" in Fig. 11 b) (the result of "f(x0)") is an activated object name which is the representation of a closure. The destination node name field of the value is node name "fx" in Fig. 11 a). Similarly, the value sent back to "L2" in Fig. 11 b) is also an activated object name whose destination node name field is "fxy" in Fig. 11 a).

Details of the closure computation process with the DCSB model are shown using a simpler example. Consider evaluating the block

```
val = { add= ^[[x] ^[[y] x+y ] ] ;
        inc=add(1) ; return inc(4)*inc(3) }
```

The evaluation proceeds as follows:

```
inc = add(1)
    = ^[[x] ^[[y] x+y ] (1)
    = ^[[y] 1+y ]
```

Value "inc" is a closure and is applied to an actual parameter "4".

```
Thus,  inc(4) = ^[[y] 1+y ] (4)
              = 1+4
              = 5
```

Figure 12 shows the computation process for the above evaluation with the DCSB model.

(1)-(2) An activated object name [⊥ | add] is given to the

"call" instruction. "⊥" means that the function "add" has no free variables. A new color "c" (⊥ ⊗ c) is allocated dynamically as the result of "call" instruction execution.

(3)-(4) A constant "1" is sent to the body of the function "add".

(5) The destination node name for a result token is sent.

(6) A free variable "x" in the function body of "inc" is bound to the actual parameter of the function "add".

(7)-(8) A closure, including a function name "inc" and an environment color "c", is returned as the result of add(1).

(9) A new color "d" (c ⊗ d) is allocated dynamically as the result of "call" instruction execution.

(10)-(12) An actual parameter and a result destination node name are sent to the function body of "inc".

(13) The token (6) and the token (11) are matched at the generation bridging operator "+", and a result token is produced.

(14) The result token is returned, and the evaluation of inc(4) is finished.

Similarly, inc(3) is evaluated, repeating steps from (9) to (14). Evaluation steps from (1) to (8) are shared and are not repeated.

As shown above, an environment is specified by a set of colors having total ordering. The order corresponds to the binding order of free variables, and static binding allows us to analyze it at compilation time. The analyzing algorithm is closely discussed in Section 4.1. The environment is represented by the minimum color of the set, and a closure is specified by an activated object name. A function application is executed using a color prime under the color sub-field of the activated object name.

In this way, the DCSB model has general computational power to evaluate a function application in an environment specified by a closure, on condition that static binding is used. The CT model lacks this power.

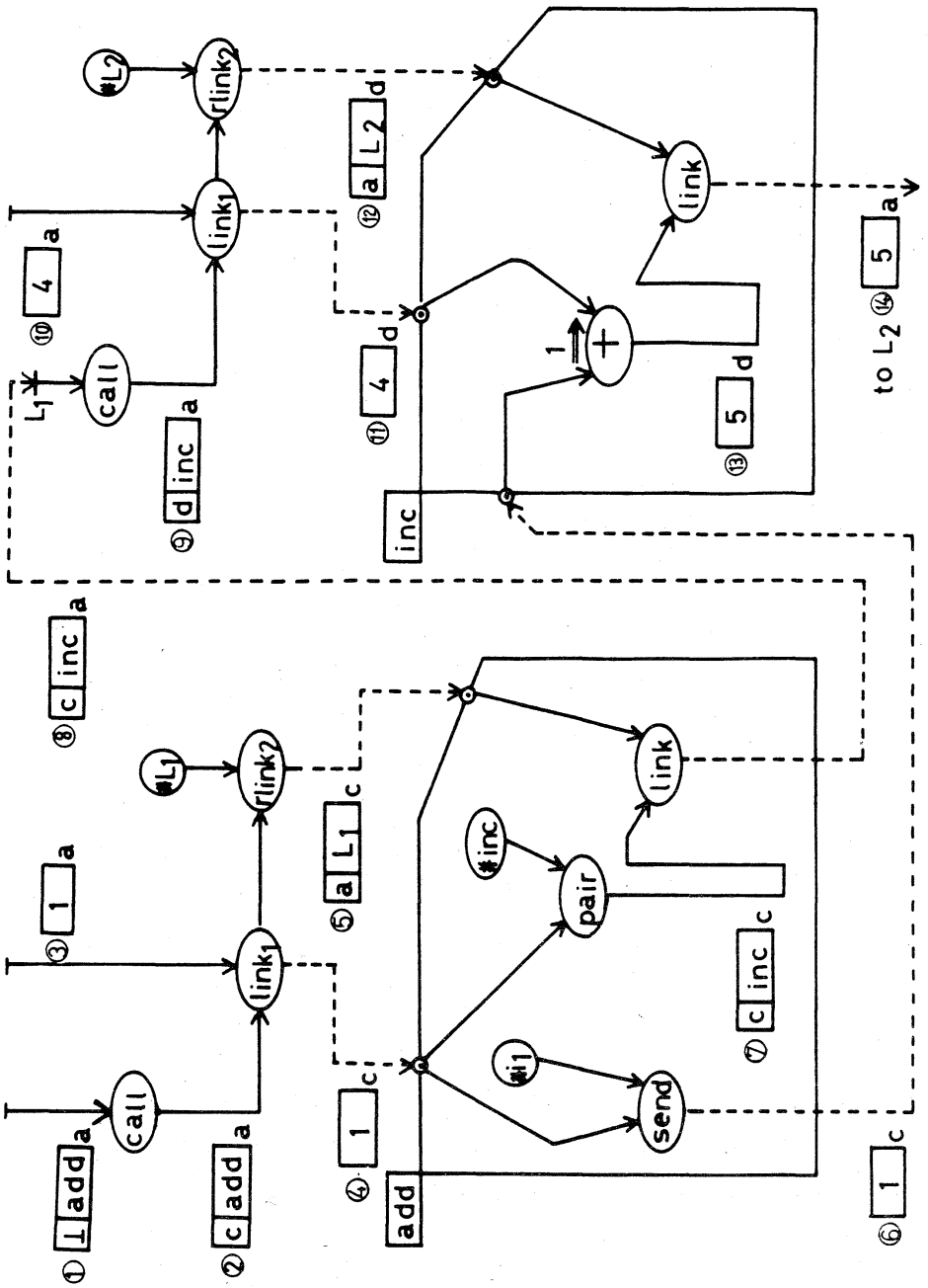


Fig. 12. Computation example with the DCSB model

3.5 Partial Application with the DCSB Model

Given a function;

```
f = ^[[x,y,z] g(x,y,z) ]
```

If variable "x" is fixed to "x0", we can obtain a function of the remaining formal parameters "y" and "z";

```
fx = ^[[y,z] g(x0,y,z) ]
```

As shown above, applying a function "f" to a subset of the parameters and computing a function taking the rest of the parameters is called a partial application.

In the above example, further partial application is possible. If variable "y" is fixed to "y0", then the following function "fxy" can be computed.

```
fxy = ^[[z] g(x0,y0,z) ]
```

Another partial application is possible for "fx". If variable "z" is fixed to "z0", we can obtain "fxz", where

```
fxz = ^[[y] g(x0,y,z0) ]
```

The partial application sequence for the DCSB model is shown in Fig. 13, where the application order is "x", "y" and "z" in sequence.

The DCSB model can execute partial application only if the partial application sequence is statically determined at compilation time. For example, consider a function application f(x0,y0) where f = ^[[x,y] x-y]. There are the following three partial application sequences.

a) apply "f" to "x0" and "y0" simultaneously.

b) partially apply "f" to "x0", then "y0".

c) partially apply "f" to "y0", then "x0".

Although all of the above function applications compute "x0-y0", the dataflow graphs for each application sequence are different, as shown in Fig. 14. Therefore, it is impossible to share dataflow graphs having different partial application sequences.

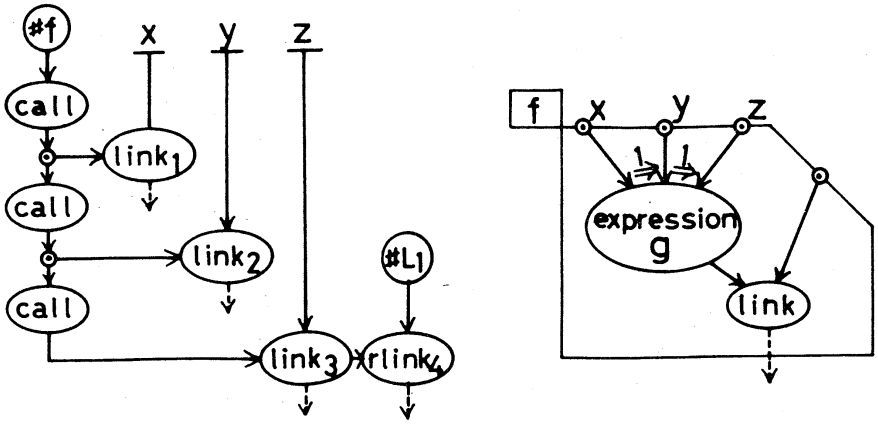
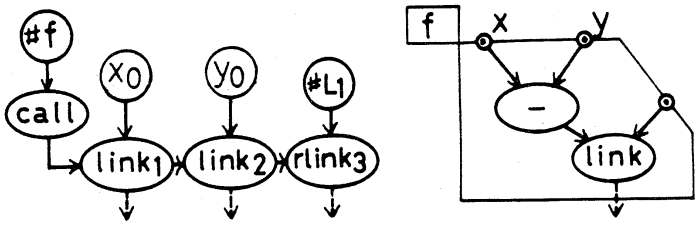
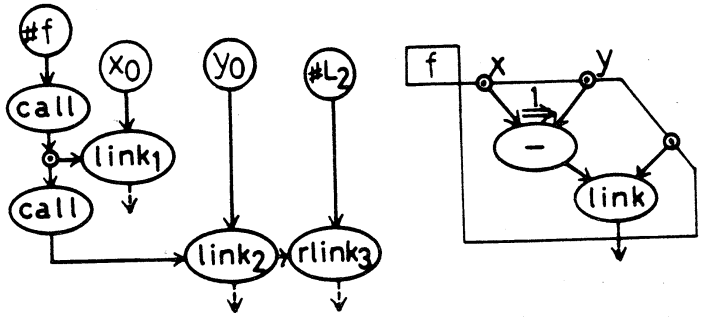


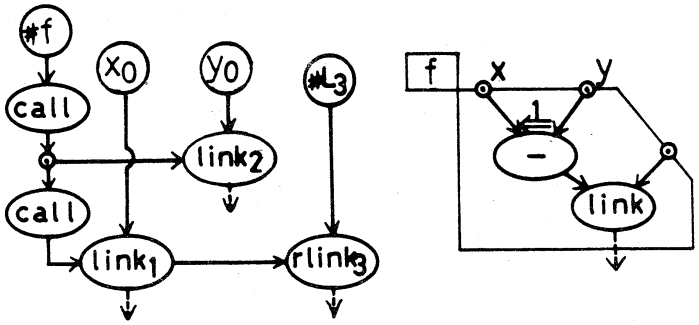
Fig. 13. Dataflow graphs for partial application with the DCSB model



a) apply "f" to "x0" and "y0" simultaneously.



b) partially apply "f" to "x0", then "y0".



c) partially apply "f" to "y0", then "x0".

Fig. 14. Dataflow graphs for f(x0,y0), where f = ^[[x,y] x-y]

Notice that the difference in the graphs in Fig. 14 is localized to the generation difference only. Since the DCDB model can determine this difference dynamically, the DCDB model can share the function graph even if partial application sequence is different. From this perspective, the DCDB model has a more general code sharing ability than the DCSB model.

3.6 Partial Simplification with the DCSB Model

VALID as well as other recent applicative programming languages, such as SASL[1], permits functions to be non-strict. A non-strict function is a function which can return a result, even if some of the parameters are undefined. Since any composition of strict functions gives a strict function, non-strictness of the function derives from non-strictness of primitive functions. A typical non-strict primitive function is an "if-then-else-fi" function. Consider evaluating a function application f(1) where

```
f = ^[[x] { g = ^[[y] if x>0 then x else y fi + x ] ;
            return g(2) + g(3) } ]

f(1) = ^[[x] { g = ^[[y] if x>0 then x else y fi + x ] ;
               return g(2) + g(3) } ] (1)
     = { g = ^[[y] if 1>0 then 1 else y fi + 1 ] ;
         return g(2) + g(3) }
     = { g = ^[[y] 1 + 1 ] return g(2) + g(3) }
     = { g = ^[[y] 2 ] return g(2) + g(3) }
     = ^[[y] 2 ] (2) + ^[[y] 2 ] (3)
     = 2 + 2
     = 4
```

In the above computation, "if 1>0 then 1 else y fi" is simplified to "1" at the time "y" is not bound (i.e. "y" is undefined). Therefore, a further computation from "1 + 1" to "2" can be shared among function applications g(2) and g(3). As shown above, simplifying an expression having undefined variables is called a partial simplification.

The string reduction mentioned above has a general partial

simplification ability. On the other hand, the DCSB model requires that the color of the result token be statically determined. Therefore, the model can have only restricted partial application power. Consider the expression evaluation "if x>0 then x else y fi" shown above, where the colors of "x" and "y" are "a" and "b" (a > b), respectively. If "x" equals 1, we can obtain a value for the expression without "y". Therefore, the result color may be "a". However, if "x" equals -1, value "y" is required and the result color should be min(a,b)=b. To avoid such a non-determinancy, the result color must always be "b". Therefore, with the DCSB model, it is impossible to execute a simplification corresponding to the reduction from "if 1>0 then 1 else y fi" to "1", unless "y" is bound and is allocated color "b" by a function application of "g".

The DCDB model has no such constraints. From this perspective, the DCSB model has less partial simplification power than the DCDB model.

4. Code Generation Method for the DCSB Model

One difficulty encountered with the DCSB model is that this model requires the generation difference on each node to be determined statically at compilation time.

In Section 4.1, a notion of a relative generation for an expression is introduced. Section 4.2 gives a description of code generation for function applications. In Section 4.3, code generation for simplifications is discussed.

4.1 Relative Generations for Expressions

Each expression of an applicative programming language has an environment in which the expression is to be evaluated. Such environments have a tree structure corresponding to the binding process of free variables and formal parameters [11].

Now, we would like to introduce a notion of relative generation, which corresponds to the depth of access environments[11].

Consider the applicative programming language described in Section 3.1. Function "r" can be defined as follows, where r("<expression>") gives a relative generation for the expression:

(1) If "x0" is a constant, then r("x0") = 0
For example, r("1") = 0

(2) If "e" is an expression of "x1", "x2",..., "xn", then
r("e") = max (r("x1"), r("x2"), ... , r("xn"))
For example, if r("x")=1 and r("y")=2, then
r("x+y+3") = 2

(3) If "f" is a function definition containing no free variables, then
r("f") = 0
For example, r("^[[x,y] x+y]") = 0

(4) If "f" is a function definition containing free variables "x1", "x2",...,"xn", then
r("f") = max (r("x1"), r("x2"), ... , r("xn"))
For example, if r("x")=1 and r("y")=2, then
r("^[[z] x+y+z]") = 2

(5) If "f" is a function definition with formal parameters "x1", "x2",...,"xn" and r("f")=n, then
r("x1") = r("x2") = ... = r ("xn") = n+1
For example, r("x") in a function definition ^[[x] x+1] is equal to 1.

(6) If "a" is an application of function "f" with actual parameters
"e1", "e2",...,"en" (i.e. a=f(e1,e2,...,en)), then
r("a") = max (r("f"), r("e1"), ... , r("en"))
For example, if r("a")=1 and r("b")=0, then
r("^[[x,y] x+y] (a,b)")
= max (r("^[[x,y] x+y]"), r("a"), r("b"))
From rule (3),
r("^[[x,y] x+y]") = 0
Therefore,

```
          r("^[[x,y] x+y ] (a,b)")
        = max ( 0, 1, 0 ) = 1
```

(7) If "v" is a left side identifier of a value definition "v = e" where "e" is an expression and r("e")=n, then

```
r("v") = n
```

For example, if r("x")=1 and "v" is defined as "v = x+3", then

```
r("v") = 1
```

Applying rules (1) - (7) recursively, a relative generation for each identifier can be assigned.

For example,

```
f = ^[[x] { a=p(x) ; h=^[[z] p(a)+q(x,z)+1 ] ;
            return h(x) } ]
```

where functions "p" and "q" have no free variables.

Relative generations can be computed as follows:

```
(a) r("1") = r("p") = r("q") = 0  ( from rules (1) and (3) )
(b) r("^[[x] { a=p(x) ; h=^[[z] p(a)+q(x,z)+1 ] ;
               return h(x) } ]") = 0
                                  ( from (a) and rule (4) )
(c) r("f") = 0                    ( from (b) and rule (7) )
(d) r("x") = 1                    ( from (b) and rule (5) )
(e) r("p(x)") = 1                 ( from (a), (d) and rule (6) )
(f) r("a") = 1                    ( from (e) and rule (7) )
(g) r("^[[z] p(a)+q(x,z) ]") = 1  ( from (a), (d) and rule (4) )
(h) r("h") = 1                    ( from (g) and rule (7) )
(i) r("z") = 2                    ( from (g) and rule (5) )
(j) r("q(x,z)") = 2               ( from (i) and rule (6) )
(k) r("h(x)") = 1                 ( from (d), (h) and rule (6) )
```

If a function "f" is defined recursively, computation of r("f") requires a special treatment. Consider

```
fact = ^[[n] if n=0 then 1 else n * fact(n-1) fi ]
```

To get r("fact"), we must get

```
r("^[[x] if x=0 then 1 else x * fact(x-1) fi ]")
```

which requires that r("fact") be defined. This circular

reference can be resolved by introducing an unknown quantity to the above rules. For example, assume

```
n = r("fact")
```

Applying the above rules, we can get the following equation.

```
n = max(0,n)
```

The minimum value satisfying the above equation should be used as "n". Therefore, r("fact") = 0.
This method can also be applied to mutually recursive functions.

4.2 Code Generation for Function Application

As discussed in Sections 3.4 and 3.5, major features not available in the CT model are the capability of processing a closure and the partial application feature.

(1) Closure

The applicative language discussed in this paper uses a static binding rule. Therefore, the environment where free variables are bound can be computed from the source program. Generation differences of operation nodes for matching a free variable "a" and a bound variable "b" are set to "r(b) - r(a)", where "r" is the function defined in the previous section.

For example, consider function "f", where

```
f  = ^[[x] { a=x+1 ; g=^[[y] a+y ] ; return g(g(x)) } ]
```

A free variable "a" in the function body of "g" is bound to the value definition "a=x+1" in a surrounding block. Relative generations can be computed for "a" and "y" as follows, using the rules presented in Section 4.1.

```
r("f") = 0          ( from rule (3) )
r("x") = 1          ( from rule (5) )
r("a") = 1          ( from rules (2) and (7) )
r("g") = 1          ( from rule (4) )
r("y") = 2          ( from rule (5) )
```

Therefore, the generation difference for matching "a" and "y" is

$$r("y") - r("a") = 1$$

We can obtain a dataflow graph as shown in Fig. 15 for the function definition "f".

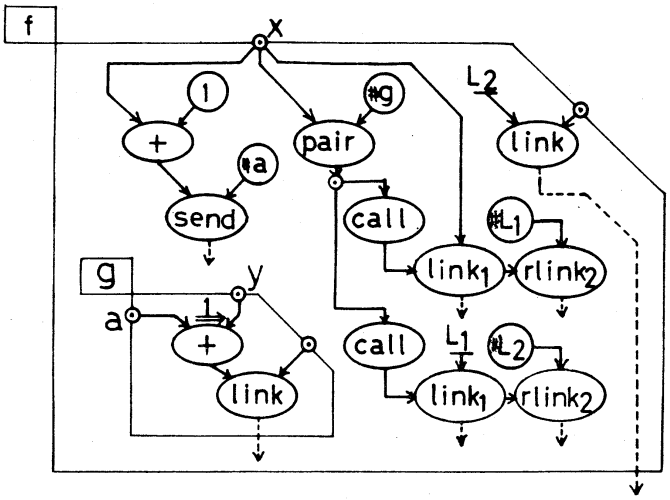


Fig. 15. Dataflow graph for function definition "f"
where f = ^[[x] { a=x+1 ; g=^[[y] a+y] ; return g(g(x)) }]

(2) Partial application

Given function definitions,

```
p = ^[[f] { q = ^[[x] g(f,x) ] ; return q(0) + q(1) } ] ;
g = ^[[f,x] f(f(x)) ]
```

The relative generations can be computed as follows:

$$r("p") = r("g") = 0 ;$$
$$r("f") = r("q") = 1 ;$$
$$r("x") = 2$$

Consider the function application "g(f,x)". The actual parameters "f" and "x" have relative generations "1" and "2", respectively. Relative generations corresponds to the order of binding in the function application process. Therefore, if the actual parameters of a function have different relative generations, partial application is possible. In the above example, function "g" can be partially applied to the actual parameter "f" initially. Then, the result function can be

applied to the remaining actual parameter "x". This process can also be simulated by "Currying"[1] functions as follows:

```
p = ^[[f] { q = ^[[x] g'(f)(x) ] ; return q(0) + q(1) } ] ;
g'= ^[[f] ^[[x] f(f(x)) ] ] ;
```

where the function "g'" is a compiler generated function from the function definition of "g". The partial application approach is adopted, since it preserves the symmetricity of the parameters, which permits the sharing of graphs in the DCDB model as mentioned in Section 3.5. The dataflow graphs for function "p" and "g" are shown in Fig. 16.

4.3 Code Generation for Simplification

Almost the same method as described in the previous section can be applied to the code generation for primitive functions. The only exception is the code for non-strict primitives, such as the "if-then-else-fi" function, discussed in Section 3.6. Assume

```
e = if x-1>y then y else z fi ;
```

and `r("x") = a ; r("y") = b ; r("z") = c ;` (a < b < c)

`c - b = m ; b - a = n` (m, n > 0)

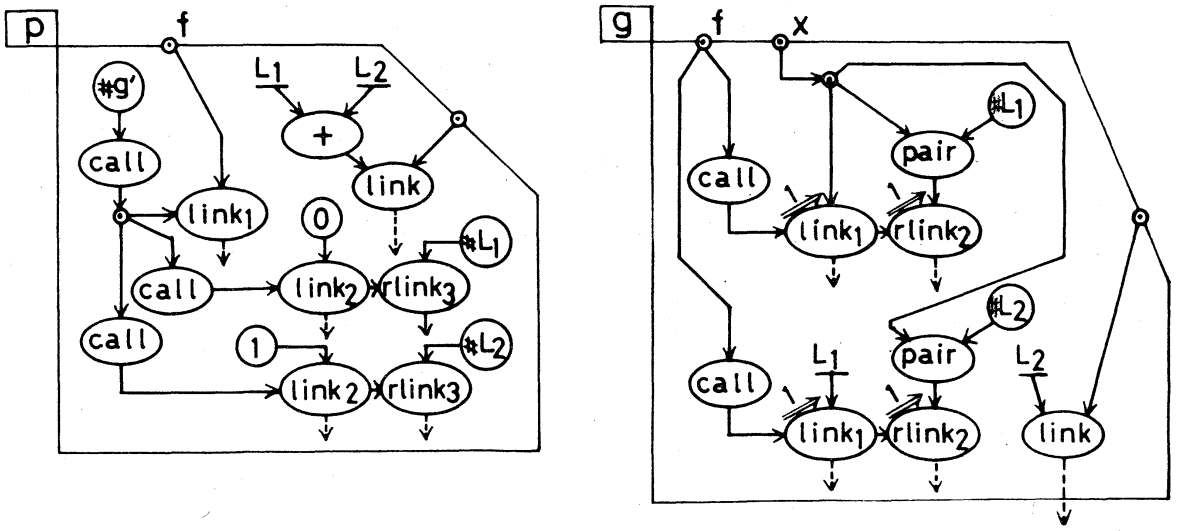


Fig. 16. Dataflow graph for function definitions "p" and "g"
where p = ^[[f] { q = ^[[x] g(f,x)] ; return q(0) + q(1) }] ;
g = ^[[f,x] f(f(x))]

Then, the following relative generations can be obtained by applying the rules in Section 4.1.

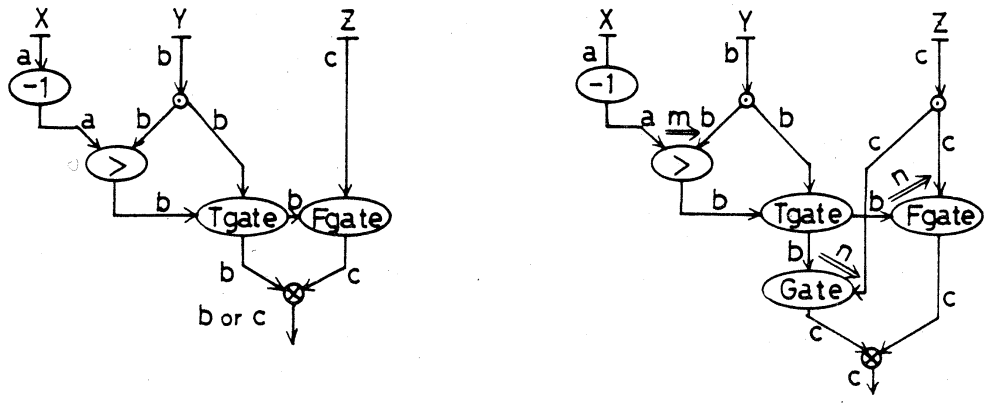
```
r("x-1") = a ;
r("x-1>y") = b ;
r("if x-1>y then y else z fi") = c ;
```

A dataflow graph for an expression "e" is shown in Fig. 17 a), where symbols "a", "b" and "c" represent the relative generations of the corresponding arcs.

However, the relative generation of "e" cannot be determined uniquely at the merge node, where relative generations of two input arcs are "b" and "c", respectively. Such a condition is called collision of the relative generations. This occurs because the implementation in Fig. 17 a) actually realizes the non-strictness of the "if-then-else-fi" function. The collision cannot be permitted since the DCSB model requires relative generations to be uniquely determined statically.

We can easily resolve the collision using dummy "gate"[3] instructions, as shown in Fig. 17 b). This is a strict implementation of an if-then-else-fi function. This type of transformation is called collision resolution.

As discussed in Section 3.6, the DCSB model has only restricted partial simplification power. This is because of the necessity



a) Before collision resolution b) After collision resolution

Fig. 17. Collision resolution

of these collision resolution in the DCSB model.

5. Conclusion

This paper has presented a new dataflow computation model, called Generation Bridging Operator (GBO) model. The main features of GBO model are as follows:

(1) The use of a partially ordered color set, called a tree structured set, as well as newly defined firing rules extended from those of the colored token (CT) model.

(2) The ability to process a closure (a pair composed of a function and an environment), which is essential to higher-order function evaluation. This model also has computational power for partial computation.

This paper has concentrated on one type of the GBO model, named Dynamic Coloring Static Bridging (DCSB) model. This paper has shown the ability and the limitation of the DCSB model in terms of closure processing, partial application and partial simplification. Furthermore, this paper clarified a dataflow graph generation method for the DCSB model by describing the essential difference in code generation between the DCSB model and the CT model.

In order to clarify the effectiveness of the GBO model, the following feasibility studies are required.

(1) The DCDB model formalization and the study of its ability and limitation.

(2) Clarification of hardware system implementation based on the GBO model.

(3) Comparison of the GBO model with the reduction model, in terms of computational power and hardware implementation difficulty.

(4) Discussion of partial computation efficiency with the GBO model by simulating program execution with both the CT model and the GBO model.

[Acknowledgements]

The authors would like to thank Dr. Noriyoshi Kuroyanagi, director of the Research Division at Musashino Electrical Laboratory, and Dr. Katsuji Tsukamoto, director of the eighth research section, for their guidance and encouragement. They also wish to thank the members of the architecture research group in the eighth research section, for their helpful discussion. The authors would also like to express their thanks to Atsuko Nanmoku for her work on the figure drawings.

References

[1] Turner,D.A.: "A New Implementation Technique for Applicative Languages," Software Practice and Experience, Vol. 9, 1979, pp. 31-49.

[2] Keller,R.M.: "FEL (Function-Equation Language) Programmer's Guide," AMPS Technical Memorandum No. 7, University of Utah, April 1982.

[3] Amamiya,M., Hasegawa,R. and Mikami,H.: "List processing with a Data Flow Machine," Lecture Notes in Computer Science, RIMS Symposia on Software Science and Engineering, Springer-Verlag, 1982.

[4] Arvind and Kathail,V.: "A Multiple Processor Dataflow Machine That Supports Generalized Procedures," Proceedings of the 8th Annual Symposium on Computer Architecture, May 1981, pp. 291-302.

[5] Gurd,J. and Watson,I.: "Data Driven System for High Speed Parallel Computing (1 & 2)," Computer Design, Vol. 9, No. 6 & 7, June & July 1980, pp. 91-100 & 97-106.

[6] Takahashi,N. and Amamiya,M.: "A Data Flow Processor Array System : Design and Analysis," Proceedings of the 10th Annual Symposium on Computer Architecture, June 1983, pp. 243-250.

[7] Amamiya,M., Hasegawa,R., Nakamura,O. and Mikami,H.: " A list-processing-oriented data flow machine architecture," Proceedings of the 1982 National Computer Conference, AFIPS, 1982, pp. 143-151.

[8] Keller,R.M., Lindstrom,G. and Patil,S.: "A Loosely Coupled Applicative Multiprocessing System," Proceedings of the 1979 National Computer Conference, AFIPS, Vol. 49, 1979, pp. 613-622.

[9] Darlington,J. and Reeve,M.: "ALICE : A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages," Proceedings of the 1981 ACM/MIT Conference on Functional Programming Language and Computer Architecture, 1981, pp. 65-75.

[10] Henderson,P.: "Functional Programming, Application and Implementation," Prentice-Hall, 1980.

[11] Allen,J.: "Anatomy of LISP," McGraw-Hill, 1978.

[12] Ershov,A.P.: "Mixed Computation : Potential Application and Problems for Study," Theoretical Computer Science 18, 1982, pp. 41-67.

[13] Backus,J.: "Can Programming be Liberated from the von Noumann Style? A Functionall Style and its Algebra of Programs," Comm. ACM, Vol. 21, No. 8, 1978, pp.613-641.